Enabling SMT for Real–Time Embedded Systems

Article · January 2004

6 authors, including:

Francisco J Cazorla
Barcelona Supercomputing Center
199 PUBLICATIONS **2,716** CITATIONS

SEE PROFILE

Rizos Sakellariou
The University of Manchester
171 PUBLICATIONS **3,498** CITATIONS

SEE PROFILE

Enrique Fernandez
Universidad de Las Palmas de Gran Canaria
23 PUBLICATIONS **481** CITATIONS

SEE PROFILE

Mateo Valero
Universitat Politècnica de Catalunya
613 PUBLICATIONS **8,400** CITATIONS

SEE PROFILE

Some of the authors of this publication are also working on these related projects:

12th Workshop on Workflows in Support of Large-Scale Science View project

AXIOM http://www.axiom-project.eu View project

ENABLING SMT FOR REAL-TIME EMBEDDED SYSTEMS

F.J. Cazorla[1], P.M.W. Knijnenburg[2], R. Sakellariou[3], E. Fernandez[4], A. Ramirez[1], M. Valero[1]

[1]DAC, UPC, Spain, {fcazorla,aramirez,mateo}@ac.upc.es
[2]LIACS, Leiden University, the Netherlands, peterk@liacs.nl
[3]University of Manchester, UK, rizos@cs.man.ac.uk
[4]University of Las Palmas de Gran Canaria, Spain, efernandez@dis.ulpgc.es

ABSTRACT

In order to deal with real time constraints, current embedded processors are usually simple in-order processors with no speculation capabilities to ensure that execution times of applications are predictable. However, embedded systems require ever more compute power and the trend is that they will become as complex as current high performance systems. SMTs are viable candidates for future high performance embedded processors, because of their good cost/performance trade-off. However, current SMTs exhibit unpredictable performance. Hence, the SMT hardware needs to be adapted in order to meet real time constraints.

This paper is a first step toward the use of high performance SMT processors in future real time systems. We present a novel collaboration between OS and SMT processors that entails that the OS exercises control over how resources are shared inside the processor. We illustrate this collaboration by a mechanism in which the OS cooperates with the SMT hardware to guarantee that a given thread runs at a specific speed, enabling SMT for real-time systems.

1. INTRODUCTION

To deal with real time constraints, current embedded processors are usually simple in-order processors with no speculation capabilities. However, embedded systems are required to host more and more complex applications and have higher and higher data throughput rates. In order to meet these growing demands, future embedded processors will resemble current high performance processors. For example, the new Philips TriMedia already has a deep pipeline, an L1 and L2 cache, and branch prediction [4]. Simultaneous Multithreaded (SMT) architectures [9][10], are viable candidates for future high performance embedded processors, because of their good cost/performance trade-off [5]. There already exists an embedded SMT communications platform on the market, called META [5]. In an SMT, several threads are running together, sharing resources at the micro-architectural level, in order to increase throughput. The front end of a superscalar is adapted in order to fetch from several threads while the back end is shared among the threads. A *fetch policy* decides from which threads instructions are fetched, thereby implicitly determining the way processor resources, like rename registers or IQ entries, are allocated to the threads. The common characteristic of many current fetch policies is that they attempt to improve a-priori established metrics like throughput [9] or fairness [6]. However, a problem with all these policies is that the performance of a thread in a workload is unpredictable. For example, Figure 1 shows the IPC of the `gzip` benchmark when it is run alone (full speed) and when it is run with other threads using two different fetch policies, *icount* [9] and *flush* [8]. Its IPC varies considerably, depending on the fetch policy as well as characteristics of the other threads running in the workload. This poses problems for the suitability of SMT processors in the context of real-time systems. Thus, if we want to be able to provide real time capabilities for an SMT processor, current approaches to resource management by means of instruction fetch policies are no longer adequate. Hence, a new paradigm for resource management inside SMT processors is required.

The key issue is that in the traditional collaboration between OS and SMT, the OS only assembles the workload while it is the processor that decides how to execute this workload, implicitly by means of its fetch policy. Hence, part of the traditional responsibility of the OS has "disappeared" into the processor. One consequence is that the OS may not be able to guarantee time constraints even though the processor has sufficient resources to do so. To deal with this situation, the OS should be able to exercise more control over how threads are executed and how they share the processor's internal resources.

In this paper, we discuss our philosophy behind a novel collaboration between OS and SMT in which the SMT processor provides 'levers' through which the OS can fine tune the internal operation of the processor to achieve certain requirements. We want to reserve resources inside the SMT
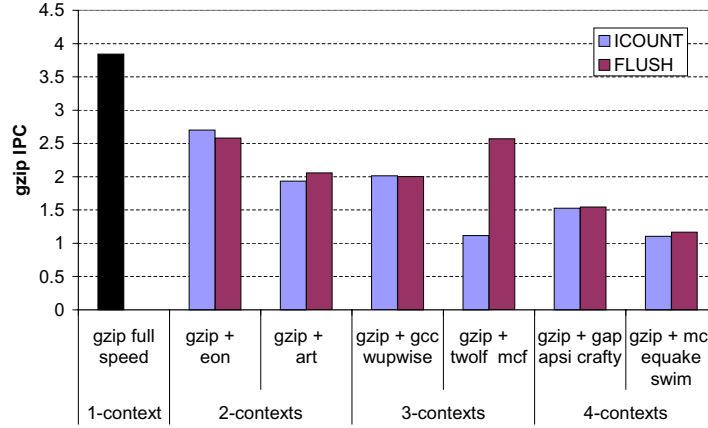
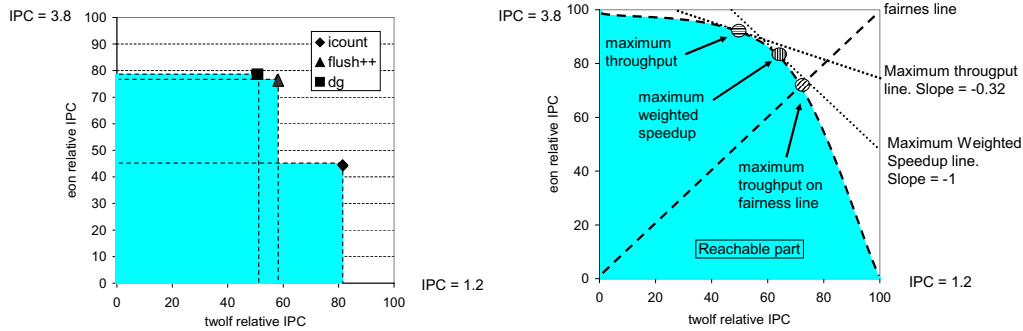Figure 1: IPC of `gzip` for different contexts and different fetch policies

Figure 2: (a) QoS space for three fetch policies; (b) important QoS points and areas

processor in order to guarantee certain requirements for executing a workload. We show the feasibility of this approach by a mechanism to achieve a given percentage of the full speed of a designated High Priority Thread. This enables the use of out-of-order, high performance SMT processor in embedded environments.

This paper is structured as follows. In Section 2 we describe our novel approach to the collaboration between OS and SMT. In Section 3 we discuss our mechanism to enable such collaboration. Finally, Section 4 is devoted to conclusions and future directions.

2. COLLABORATION BETWEEN OS AND SMT

In this paper, we approach OS/SMT collaboration as *Quality of Service (QoS)* management. This approach is inspired by QoS in networks in which processes are given guarantees about bandwidth, throughput, or other services. Analogously, in an SMT resources can be reserved for threads guaranteeing a required performance. We observe that on an SMT processor, each thread reaches a certain percentage of the speed it would achieve when running alone on the machine. Hence, for a given workload consisting of N applications and a given instruction fetch policy, these fractions give rise to a point in an N-dimensional space, called the *QoS space*. For example, Figure 2(a) shows the QoS space for two threads, `eon` and `twolf`, as could be obtained for the Pentium4 or the Power5. In this figure, both x- and y-axis span from 0 to 100%. We have used three fetch policies: *icount* [9], *flush++* [1], and data gating (*dg*) [3]. Theoretically it is possible to reach any point in the shaded area be-

low these points by judiciously inserting empty fetch cycles. Hence, this shaded area is called the *reachable part* of the space for the given fetch policies. In Figure 2(b), the dashed curve indicates points that intuitively could be reached using some fetch and resource allocation policy. Obviously, by assigning all fetch slots and resources to one thread, we reach 100% of its full speed. Conversely, it is impossible to reach 100% of the speed of each application at the same time since they have to share resources.

In Figure 2(b) we see that the representation of the QoS space also provides an easy way to visualize other metrics used in the literature. Points of equal weighted speedup, as defined in [7], lie on the same line that is perpendicular to the bottom-left top-right diagonal. In Figure 2(b) the point with maximum weighted speedup in the reachable part is indicated. Similarly, points of equal throughput lie also on a single line whose slope is determined by the ratio of the maximum IPCs of each thread (in this case, $-1.2/3.8 = -0.32$). Such a point with maximum throughput is also indicated in the figure. Finally, points near the bottom-left top-right diagonal indicate fairness, in the sense that each thread achieves the same proportion of its maximum IPC. In either case, maximum values lie on those lines that have a maximum distance from the origin.

Each point or area in the reachable subspace entails a number of properties of the execution of the applications: maximum throughput; fairness; real-time constraints; power requirements; a guarantee, say 70%, of the maximum IPC for a given thread; any combination of the above, etc. Put differently, each point or area in the space represents a so-

lution to a *QoS requirement*. It is the responsibility of the OS to select a workload and a QoS requirement and it is the responsibility of the processor to provide the levers to enable the OS to pose such requirements. To implement such levers, we consider the SMT as having a collection of sharable resources and add mechanisms to control how these resources are actually shared. These mechanisms include prioritizing instruction fetch for particular threads, reserving parts of the resources like instruction or load/store queue entries, prioritizing issue, etc. The OS, knowing the needs of applications, can exploit these levers to navigate through the QoS space. This solution should be parameterized and maximize the reachable part of the space so that it is generally usable and provides opportunities for fine tuning the machine for arbitrary workloads and QoS requirements.

In this paper, we present a novel resource managment mechanism that enables the processor to execute a designated High Priority Thread at a given percentage of its full speed, that is, the speed it would obtain when running alone on the machine. At the same time, it maximizes the throughput of the Low Priority Threads. This mechanism enables the OS to deal with Worst Case Execution Times and hence enbles the use of SMT processors in real time systems. This novel mechanism is dynamic and tries to navigate toward a required area in the QoS space.

3. QOS BY RESOURCE ALLOCATION

In this section, we discuss a novel dynamic mechanism that is capable of solving a QoS requirement that requires to run a specific job at a given percentage of its full speed, that is, of the IPC it would have when executed on the machine on its own. By way of example, we show that we can achieve 70% of the full speed of `gzip` while it is running in several contexts, while at the same time maximizing as much as possible the performance of the other threads in the workload.

3.1 Methodology

We assume a fairly standard 4-context SMT configuration: our machine can fetch up to 8 instructions from up to 2 threads each cycle. It has 6 integer, 3 FP, and 4 load/store functional units and 32-entry deep integer, load/store and FP IQs. There are 320 shared physical registers shared for all threads. Each thread has its own 256-entry reorder buffer. We use a 2 level cache hierarchy with separate 32K, 4-way data and instruction caches and a unified 512KB 8-way L2 cache. The latency from L2 to L1 is 10 cycles, and from memory to L2 100 cycles. We use a trace driven SMT simulator, based on SMTSIM provided by Tullsen [9]. It consists of our own front-end that reads a trace file and a modified version of SMTSIM's back-end. We run 300 million most representative instructions for each benchmark.

In our experiment, we consider contexts of 2, 3, and 4 threads. We consider two types of threads: threads that exhibit a high number of L2 misses, called *Memory Bounded* (MB) threads. These threads have a low full speed. Secondly, threads that exhibit good memory behavior and have a high full speed, called *ILP threads*. We always use the ILP thread `gzip` as High Priority Thread (HPT). The Low Priority Threads (LPTs) are either all ILP or MB. They are denoted by In and Mn, respectively, where n is the number of LPTs. For example, I1 denotes the workload `gzip` and `eon`, and M2 the workload composed of `gzip`, `mcf`, and `twolf` (see Figure 4).

3.2 Dynamic resource allocation

We propose a mechanism that provides control over the execution speed of a designated High Priority Thread by dynamically allocating resources to it. It ensures that the HPT runs at a given target IPC that represents $X\%$ of its IPC

when it would run alone on the machine. At the same time, we want to give best effort to the remaining Low Priority Threads and maximize their throughput as well.

The basis of our mechanism rests on the observation that in order to realize $X\%$ of the overall IPC for a given job, it is sufficient to realize $X\%$ of the maximum possible IPC *at every instant* through the execution of that job. We employ two phases, discussed in more detail in [2], that are executed in alternate fashion.

- During the *sample phase*, all shared resources are given to the HPT and LPTs are temporarily stopped. As a result, we obtain an estimate of the full speed of the HPT during this phase, called the *local IPC*. In order to counteract thread interference, this phase is divided in a *warmup phase* of 50,000 cycles and an *actual sample phase* of 10,000 cycles.

- During the *tune phase*, our mechanism dynamically varies the amount of resources given to the HPT in order to achieve a *target IPC* that is given by the local IPC computed in the last sample period times the required percentage given by the OS. The resources considered in this paper are rename registers, instruction and load/store queue entries, and ways in the 8-way set associative L2 cache. We tune resource allocation every 15,000 cycles for a period of 1.2 million cycles.

3.3 Results

In Figure 3, we show the achieved percentage of the full speed of `gzip` and the full speed of the Low Priority Threads as points in the QoS space. The full speed of the LPTs is the speed that they would obtain if they were executed as a single workload using *flush*. From this figure, we observe that the *icount* and *flush* policies are scattered through the QoS space. Instead, our QoS mechanism always achieves 70% of the full speed of `gzip` or slightly more. This shows that we can isolate the execution of `gzip` from the other threads and hence enable real time constraints on an SMT processor. In [2] we have shown that we can reach arbitrary percentages for a large collection of workloads with the same accuracy.

Figure 4 shows that, on average, the total throughput of our QoS mechanism does not degrade compared to either *icount* or *flush*. The horizontal lines in the bars denote the IPC values of the HPT. We see that in all cases the throughput of the LPTs is lower when using our QoS mechanism than for either *icount* or *flush*. This is because we need to reserve many resources for the HPT in order to reach a high percentage of its full speed. Hence, there is a tradeoff between throughput of the LPTs and performance predictability. For ILP workloads, the total throughput is about 10% less, also because during the sample periods the LPTs are stopped. When the ILP thread `gzip` runs in a context of MB threads, long latency loads missing in the L2 cache tend to clog the pipeline in *icount*, reducing overall IPC. This effect is less for *flush* that is designed to deal with this. Using these last two fetch policies, the MB LPTs do not have high throughput. Using our mechanism, the HPT reaches a higher, required, speed than in either *icount* or *flush*, and as a result the total throughput increases.

4. CONCLUSION

In this paper, we have approached the collaboration between OS and SMT as Quality of Service management, enabling the OS to exercise control over the execution of threads. The SMT processor provides 'levers' through which the OS can fine tune the internal operation of the processor in order to meet certain QoS requirements, expressed as points or areas in the QoS space. We have shown that it is possible to influence at will the execution of a thread in a workload in
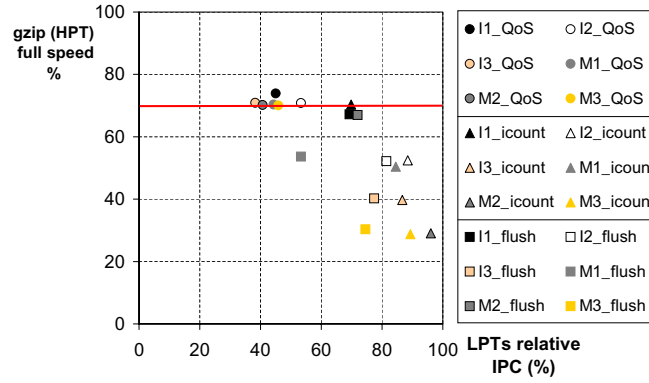
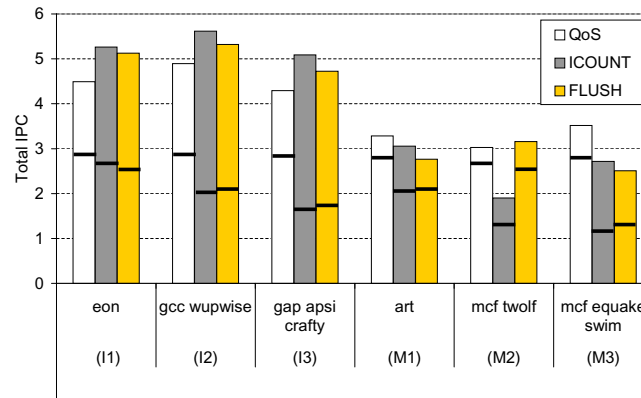Figure 3: QoS space resulting from dynamic resource allocation

Figure 4: Total IPC and IPC of the HPT for various policies

order to achieve different QoS requirements. We also have proposed a novel mechanism to deal with one of those requirements: to guarantee a job a minimum percentage of its full speed. This mechanism enables high-performance out-of-order SMT processors to deal with real-time constraints and hence renders them suitable for many types of embedded systems.

REFERENCES

[1] F. J. Cazorla, E. Fernandez, A. Ramirez, and M. Valero. Improving memory latency aware fetch policies for SMT processors. In *Proc. ISHPC*, pages 70–85, 2003.

[2] F.J. Cazorla, Peter M.W. Knijnenburg, R. Sakellariou, E. Fernandez, A. Ramirez, and M. Valero. Predictable performance in SMT processors. Technical Report UPC-DAC-2003-57, Universitat Politecnica de Catalunya, 2003.

[3] A. El-Moursy and D.H. Albonesi. Front-end policies for improved issue efficiency in SMT processors. In *Proc. HPCA*, pages 31–42, 2003.

[4] T. R. Halfhill. Philips powers up for video. *Microprocessor Report*, November 2003.

[5] M. Levy. Multithreaded technologies disclosed at MPF. *Microprocessor Report*, November 2003.

[6] K. Luo, J. Gummaraju, and M. Franklin. Balancing throughput and fairness in SMT processors. In *Proc. ISPASS*, pages 164–171, 2001.

[7] A. Snavely, D.M. Tullsen, and G. Voelker. Symbiotic job scheduling with priorities for a simultaneous multithreaded processor. In *Proc. ASPLOS*, pages 234–244, 2000.

[8] D. Tullsen and J. Brown. Handling long-latency loads in a simultaneous multithreaded processor. In *Proc. MICRO*, pages 318–327, 2001.

[9] D. Tullsen, S. Eggers, J. Emer, H. Levy, J. Lo, and R. Stamm. Exploiting choice: Instruction fetch and issue on an implementable simultaneous multithreading processor. In *Proc. ISCA*, pages 191–202, 1996.

[10] W. Yamamoto and M. Nemirovsky. Increasing superscalar performance through multistreaming. In *Proc. PACT*, pages 49–58, 1995.